Type Sensitive Application of Mutation Operators for Dynamically Typed Programs

Leonardo Bottacci
Department of Computer Science, University of Hull
Hull, HU6 7RX, UK
L.Bottaci@hull.ac.uk

Abstract—It is commonly accepted that strong typing is useful for revealing programmer errors and so the use of dynamically typed languages increases the importance of software testing. Mutation analysis is a demanding software testing criterion. Although mutation analysis has been applied to procedural languages, and object oriented languages, little work has been done on the mutation analysis of programs written in dynamically typed languages.

Mutation analysis depends on the substitution and modification of program elements. In a strongly typed language, the declared type of the mutated element, a variable or operator, can be used to avoid generating type-incorrect substitutions or modifications. In a dynamically typed language, this type information is not available and so a much greater range of mutations are potentially applicable but many of the resulting mutants are likely to be incompetent (too easily killed). This paper describes a mutation analysis method in which the definition of mutants is performed at run-time when type information is available. The type information can be used to avoid generating incompetent mutants.

Keywords—software testing, mutation analysis, dynamically typed languages, JavaScript

I. INTRODUCTION

Mutation analysis is a fault-based coverage criterion. Its origins lie in hardware testing. Hardware manufacturing is vulnerable to specific types of fault and consequently it is cost-effective to target testing at these faults. Fault-based testing, the idea of testing to eliminate specific faults, rather than a manufacturing process. In manufacturing, the set of likely faults, the fault model, is constructed by introducing faults into a design that is accepted as correct. In contrast, a correct program is rarely available. There is instead, the program written by the programmer. The key idea that makes mutation analysis feasible is that the presence of both incompetent mutants and equivalent mutants has the effect of reducing the efficiency of the testing process.

The key idea that makes mutation analysis feasible is that the set of competent programs can be approximated by making small changes to the given program under test. Such changes typically include the replacement of a program variable with some other variable or the replacement of an arithmetic or relational operator by some other compatible operator. The resulting programs are known as mutants of the given program and the modification rules are known as mutation operators.

A common misconception is that the mutation operator introduces a fault, or aims to introduce a fault, into the given program. This is not so. Since the given program may or may not be correct, a mutant may or may not be faulty. The purpose of the mutation operators is to generate the set of competent programs that a competent programmer might produce.

The mutants may be a rather crude approximation to the set of competent programs. Many of the mutants produced, although syntactically similar to the given program, may not be behaviourally similar. It is, however, relatively easy to detect such mutants. They may be type-incorrect and hence do not survive beyond the compilation stage, these are the so-called still-born mutants. Those that compile can be detected by the execution of almost any test that reaches the equivalent mutants, typically by manual inspection.

Another problem, at the opposite end of the competence scale, is that mutation operators tend to generate a small but significant number of mutants that are behaviourally indistinguishable from the given program. While, it is relatively easy to detect incompetent mutants, the problem of establishing the equivalence of two programs is undecidable in general and considerable effort may be devoted to the identification of equivalent mutants, typically by manual inspection.

The presence of both incompetent mutants and equivalent mutants has the effect of reducing the efficiency of the targeting that is the basis of fault-based testing. This is an important issue since it is recognised that mutation analysis can be costly in terms of the number of mutants that must be executed and the number that must be inspected to investigate equivalence.

In spite of these shortcomings, the set of mutants of a
given program may serve as a demanding coverage criterion against which test data can be assessed and improved. The criterion requires that all the incorrect competent programs are distinguished from the given program. Consider the execution of a test on the given program and also on some mutant. If the outputs differ, at least one of the two programs is incorrect. Such a test is informative because prior to the execution of the test, either program could have been correct or incorrect. If in contrast, the outputs do not differ, the test is uninformative. A test set is said to be mutation adequate if it is able to distinguish all non-equivalent mutants from the given program. Equivalently, a mutation adequate test set may be characterised as a test set that is maximally informative with respect to a set of mutants.

Mutation analysis has been applied largely to procedural programming languages [5]–[7], with some work on the mutation of object oriented programs [8]. In all cases, the languages have been strongly typed. Mutation analysis depends on the substitution and modification of program elements, the replacement of operators and operands in expressions is typical. In a strongly typed language, type-incorrect substitutions or modifications can be avoided since the type of the mutated element, a variable or operator, is known.

Little work, however, has been done on the mutation analysis of programs written in dynamically typed languages. In the context of computer security, the problem of cross-site scripting has been investigated [9] by applying a small number of mutation operators to JavaScript and PHP programs. The author is unaware of any other work on the mutation testing of dynamically typed programs. In such languages, type information is associated with values rather than variables. In the absence of type information for variables, a problem arises in ensuring that mutation operators do not produce “type incorrect” mutants. This paper describes a mutation analysis method in which the definition of mutants is performed at run-time when type information is available. This is beneficial in avoiding the generation of incompetent mutants. The examples are based on the JavaScript [10] language but the approach is applicable to the mutation analysis to any high-level dynamically typed language.

II. MUTATION OF DYNAMICALLY TYPED PROGRAMS

Although there is no “standard” set of mutation operators, they depend primarily on the programming language, there is a common core based on the mutation of the operands and operators in expressions. The mutation of operators, primarily the arithmetic, relational and logical operators may be performed in a dynamically typed language in much the same way as it is performed in a strongly typed language and will not be discussed further. The following discussion will cover the mutation of the operands of expressions.

A. Mutation of Simple Value Types

There are two kinds of operand mutation. Firstly, the value of the operand may be modified, as for example occurs when a number is incremented or a Boolean value is negated. In this paper, such mutations are called value mutations. Secondly, an occurrence of an operand, a variable or literal, may be replaced with another operand from the program. These are replacement mutations.

1) Value Mutations: Clearly, value mutations should be applied only to the appropriate type of value. The application of an increment mutation, for example, to a string is type-incorrect. In a strongly typed language, type information can be used to apply mutation operators statically and avoid generating mutants that are type-incorrect. This increases the efficiency of mutation analysis since a large number of incompetent mutants are never generated.

In a dynamically typed language, however the variables have no type, only values have types and values exist only at run-time. Static type information is therefore not available to restrict how mutation operators might be applied. The general approach advocated in this paper for overcoming this problem is to delay until run-time the generation of mutants that require type information. This can be done by adapting the meta-mutant design [11]. In the conventional meta-mutant, the mutants are defined at the time the meta-mutant is constructed. The mutants, may however, be defined at meta-mutant run-time. Consider value mutations. The meta-mutant may be executed with an input and at the point at which execution reaches a given variable occurrence, the type of the value held in that variable is known. At this point, the appropriate value mutations for that variable occurrence can be defined.

Deciding on the appropriate value mutations to apply to particular types is not straightforward, however. A language such as JavaScript has a large number of type conversions that are applied implicitly to operands of expressions. Take for example the typical Boolean value mutations: logical negation, constant true function and the constant false function. A direct application of these Boolean value modifications to a numeric value results in the implicit conversion of the number to a Boolean. Since every non-zero number implicitly converts to true, the logical negation of a non-zero number produces false. Strings are also implicitly converted to Boolean values with the empty string being the only “false” string. Allowing for implicit type conversions, the Boolean value mutations may be applied to Boolean values, numbers and strings.

A similar situation arises with the application of number value mutations such as increment, negation and absolute value. Boolean values are readily converted to number values. Strings that parse to numbers, convert to numbers. Operators that mutate the value of a string are of course applicable to all types since all values convert to a string. In all, the extensive implicit type conversions allow value
mutations to be applied to a wider set of values, generating larger numbers of mutants. Again, there is the danger of generating many incompetent mutants.

An alternative is to apply value mutations only to values of the specific type for which they are defined and not to values that may be converted to that type. This approach would certainly reduce the number of mutants generated. With this more restricted approach, however, there is a danger that a value mutation will not be applied to a value because it is not of the exact corresponding type although the program is such that the value is always converted to this type. As an example, consider the following code fragment in which a string value mutation is required to insert a character into the string value.

```java
s = input.getString();
if (s) {
    ...
}
```

If no Boolean mutations are applied to strings then the condition of the if-statement is not mutated and clearly it should be.

This problem can be mitigated by providing specific mutations for a given type that are designed to produce specific values after type conversion. Consider string mutations, for example. Every non-empty string converts to the Boolean true. By including a string value mutation that maps any non-empty string to the empty string, the original non-empty, i.e., "true", string that is used in a Boolean context is logically negated. To produce the same effect with an empty string, a string value mutation is required to insert a character into the empty string. Moreover, if that character is a digit then the string value may be parsed as a number and the mutation also achieves a number mutation in a context in which the string is converted to a number.

Table I shows a set of possible mutation operators for values of simple type. Note that the last three number operators in Table I perform the three Boolean mutations in a context in which the number is converted to a Boolean.

2) Replacement Mutations: In the case of replacement mutation, the type of the value held by the replacement variable should be compatible with that of the variable it replaces. It could be argued that in a dynamically typed language, any value may be assigned to any variable and hence type insensitive operand replacement is appropriate. The objective of replacement mutation, however, is to produce competent mutants and type insensitive variable replacements are likely to lead to relatively more incompetent mutants.

It is clear that type insensitive mutation operators may produce many more mutants since the set of variables with which any given variable may be replaced is potentially larger. Some replacements, in particular the replacement of objects by simple values and vice versa, will very likely lead to mutants that are semantically invalid. In the case that a simple value of one type is replaced with a simple value of another type, the outcome is not clear. The competence of such a mutant may depend on the types and values held by the variables.

In a dynamically typed language, a program may explicitly rely on variables holding values of different types. For such programs, mutants that replace a value with another of a different type may lead to valuable competent mutants. As an example of a mixed type program, consider the following simple JavaScript program.

```javascript
x = x + x;
x = x + "; /* convert x to a string
if (x) {
    return x.length;
}
else {
    return 3;
}
```

The input variable x may hold a digit, either as a number or a string, in addition, x may hold the empty string. If the input is a number then two cases apply. The program should return 1 when the input is less than 5 and return 2 when greater than or equal to 5. If the input is a string digit, it should return 2 and if the input is the empty string, it should return 3. The behaviour of the first + depends on the type of the value in x. If x is a number, the result in the number of digits in the number x + x. If x is a string, the string is concatenated with itself. In the second statement, x is concatenated with the empty string, in case x is a number, it is converted to a string. The if-statement depends on the conversion of the string value to a Boolean value.

In such a “mixed type” program, it is plausible that the various occurrences of x could be replaced by a number or
string value without necessarily generating an incompetent mutant.

In the case of a value mutation, it is necessary to know the type of the value to which it is applied. In the case of a replacement mutation, it is necessary to know not only the type of the value in the variable that is to be replaced but also the type of the value in the replacement variable. This means that replacement mutation can be performed only after all the variables have been associated with a type, or at least all the variables that can be reached by a test in the test set have been associated with a type. This can be done by arranging for all the value mutants to be executed before any replacement mutant.

In the value mutation execution phase, once execution reaches the variable to be mutated, the type of the value it holds is used to define value mutations and is also recorded for the benefit of the second, variable replacement, phase. After all the tests have been executed in the first phase, the value modification mutants of every reached mutable element will have been executed. Moreover, each reached mutable variable will have been associated with a type which can be used to define type compatible replacement mutants.

In a dynamically typed language, a variable may of course hold values of different types at different times during an execution. It is not clear at this stage the extent to which this occurs in practice and how it should influence the definition of mutations. In the majority of programs, it is expected that variables will hold values only of a single type. In the cases where more than one type is held, it is expected that the types be convertible as in the earlier program example involving digits as numbers and strings.

B. Object Mutation

1) Object Value Mutation: In addition to the values of the simple JavaScript types, i.e. Boolean, number and string, JavaScript contains the function type and the object type. Value mutation is also applicable to object values. In JavaScript, an object is a collection of named values called properties. So for example, in the program fragment

```javascript
x = {'student': 'John',
     'number': '0232'},
grade: 45};
...  
y.student = x.student;
```

The variable `x` is set to an object containing a `student` property and a `grade` property. The value of the `student` property is an object with `name` and `number` properties.

Consider the definition of a mutation operator to mutate an object value. One possibility is to mutate all the properties, at the top-level and in any sub-object. In a large object, this will lead to a large number of mutations. It is not clear how many of them would be incompetent or indeed how many would be equivalent mutants. Another possibility, and this would reduce the number of mutant generated, is to mutate only the top-level properties of the object. In the above example, this would be mutation of the `student` and `grade` properties but not the `name` and `number` properties.

In some situations, however, the mutation of only the top-level properties seems unsatisfactory. For example, consider an object containing only a single property, the value of which is a large object with many simple value properties. If object mutation were to be restricted to the top-level properties only, then, taking for example property deletion, the result would be a single mutant consisting of the deletion of a single large object. Similarly, property replacement would lead to the wholesale replacement of a large object. This seems unsatisfactory because such a large modification to an object is likely to create an incompetent mutant.

Restricting property mutation to the leaf or terminal properties of an object avoids the problem of producing a mutant with a large object modification. In the previous example, the object referenced by `x` has 3 leaf properties, `x.student.name`, `x.student.number` and `x.grade`. The value of a leaf property is, by definition, a value of a simple type to which replacement and value mutations may be applied as discussed in the previous section dealing with the mutation of simple value types. This approach towards object mutation has also been advocated by Alexander [12] in the context of the mutation of Java objects.

Since leaf property mutations are more “fine-grained” than would be obtained from the mutation of higher-level properties, it seems more likely that the resulting mutants are competent. There is the corresponding danger, of course, that if the mutations are too “fine-grained”, many of them will be equivalent. As an example of how the modification of a single property in a large object may produce an equivalent mutant, consider the JavaScript event object. This object has a number of properties which are updated in response to input events such as when a key or mouse button is pressed. The event object has a property called “shiftKey” which is set true if the shift key was pressed at the time of the event. A mutant program might set the “shiftKey” property false without otherwise modifying the event object. If the original program does not read the status of the shift key then such a mutant is equivalent.

Unlike class-based object-oriented languages, the properties of a JavaScript object may be created and deleted at runtime. This means that the identification of the properties to mutate must be made at run-time. In JavaScript, the properties of any object may be enumerated and a recursive descent of a given object can enumerate all the leaf properties.

The definition of object property value mutations is thus performed in a similar manner to the mutation of non-object values. It is essentially a repetition, for each leaf property, of the process used to mutate a non-object value.
2) Object Property Access Mutation: A property access expression, known in JavaScript as a refinement expression, is a suitable candidate for mutation, being essentially a structured name for a variable. In a property access expression such as

```javascript
o.n // can also be written o[n]
```

the variable that references the object `o` is mutated by replacing it with another variable. The replacement variable should be a variable that references an object. Recall that the variable replacement mutations are performed in the second phase using type information accumulated in the first phase.

The name component of the property access expression, `n` in the above example, may also be mutated. There are a number of ways in which this name may be mutated. In JavaScript, a relatively common error is to misspell the name of a property of an object. Recall that property access expressions cannot by checked at compile time. The consequences of such an error depend on whether the misspelt name clashes with an existing property name, in which case this is a variable replacement, or the misspelt name is new to the object. In this latter case, two further cases apply. If the new name is used in the context of an l-value (i.e. name denotes the target of an assignment), a new property is implicitly created as the target of the assignment. Such errors can be missed by even competent programmers and so this suggests the definition of a mutation operator that modifies the property name in a property access expression to a name that differs from the name of any existing property of the relevant object.

If however, the misspelt property name is used in the context of an r-value (i.e. name denotes the property of which the value is required), an “undefined” value is returned. This error is probably easier to detect but may not always be easy. The presence of the “undefined” value in an expression does not necessarily terminate the program. This suggests a mutation operator that replaces a property name with a name that results in the “undefined” value.

In the case that the property name is an expression, that expression is mutated. So for example, in the expression

```javascript
o[a[i - 2]]
```

`o` is an object and the property being accessed is defined by the expression `a[i - 2]`, i.e. the property name is an element of an array `a`. The expression `i - 2` would be subject to mutation as an arithmetic expression. Note that all the mutations of `a[i - 2]` that do not yield a valid property name for the object `o`, produce the same value, i.e. “undefined”. This may lead to a large number of incompetent mutants but since they all produce the same effect on the mutated expression, any test that distinguishes one of these mutants will distinguish all of them.

The typical JavaScript program has access to a large number of objects, many of which, e.g. the window and document objects, have a large number of properties. Unrestrained property name mutation is likely to lead to large numbers of mutants. It is not clear, however, how many of them would be competent.

III. COMMON PROGRAMMING ERRORS AND MUTATION OPERATORS

Mutation operators are not designed to emulate programming errors but rather to produce competent programs. Common programming errors may, however, inform the design of mutation operators.

Taking JavaScript as an example, it is a very permissive language. This give considerable scope for, on the one hand, concise programming, and on the other hand, programming errors. JavaScript does not, for example, check at compile-time the number of parameters supplied in a function call. At run-time, supplying a number of arguments that differs from the number of declared parameters does not lead to an invalid program. Additional arguments are ignored and missing arguments cause the corresponding parameters to take the “undefined” value.

Not withstanding the fact that programs can sometimes usefully exploit functions that are called with a variable number of arguments, clearly this is a potential source of errors. It is not clear, however, that the mutation of the function call expression, perhaps to delete arguments, is the best way to reveal such errors; especially since in many cases it is a straightforward static analysis problem to detect missing arguments. Static analysis will not detect all cases of missing arguments, however, since, in JavaScript, functions may be created at run-time.

Since variable declarations are optional in JavaScript, misspelling the name of a variable leads to the implicit declaration of a new variable. Moreover, within a function, the introduction of an undeclared variable results in the implicit declaration of a global rather than a local variable. - JavaScript has only two scopes, function and global. An obvious mutation operator is suggested by this “language feature”. The operator would remove the declaration of a variable that had a declaration and add a declaration for a variable that did not. It is not clear, however, how competent such mutants would be. In addition, it would seem more sensible to adopt a coding rule in which all variables are declared. Tools such as JSLint [13] are available to enforce such rules. Given the dubious benefit of variable declaration mutation, and the availability of tools to enforce declaration, variable declarations are not considered suitable for mutation.

IV. CONCLUSION AND FURTHER WORK

Since testing is more important for programs written in a dynamically typed language and mutation analysis is a demanding testing criterion, the combination of mutation analysis for dynamically typed programs has the potential to
be highly effective. In order that mutation operators may be applied in a type sensitive manner, however, their application to dynamically typed programs may be delayed to mutant execution time.

Clearly, empirical investigation is required, the establishment of the effectiveness of mutation analysis for dynamically typed programs. Of particular interest is the effectiveness of the run-time type information in avoiding the generation of incompetent mutants. The rate at which equivalent mutants tend to be generated is also important. It would be interesting to compare these proportions with programs written in strongly typed languages.

The role of pre-defined objects such as the Document Model in JavaScript programs has not been considered although such objects are used extensively. This is an important area for further work.

ACKNOWLEDGMENT
This work was partly supported by the EU FP7 Project ATEST2 (Grant 224442).

REFERENCES


